

What Change History Tells Us about Thread Synchronization

RUI GU, GUOLIANG JIN, LINHAI SONG, LINJIE ZHU, SHAN LU
UNIVERSITY OF WISCONSIN – MADISON, USA

Proper Thread Synchronization is Hard

Very easy to get things wrong

- Hard-to-find concurrency bugs(Data race, Atomicity Violation, Dead Lock)

Hard to debug

- Enforce complex thread scheduling between threads(GDB, LLDB)

Unexpected performance degradation

- Lock Contention

Information Beyond Bug Reports

Unreported bugs, optimizations

- Small but non-trivial concurrency bugs
- Synchronization related performance optimizations(premature, mature?)

Scattered information throughout revision history

- How is a concurrency bug introduced? Are those locks needed when the synchronization body is created?

Hidden trending information

- Is synchronization problems(correctness, performance) getting worse when software evolves?
- Do critical sections more likely to be modified when it ages?

Key Questions

- How often synchronization is an afterthought for developers?
- How many critical section related changes throughout software history? Why did developers make these changes?
- Is synchronization related problem(correctness, performance) getting worse when software evolves?
- Is over-synchronization a real problem?
- How software changes lead to synchronization-related correctness problems(Concurrency Bugs)?

Impact

Previous studies have looked at bug databases only. But not all the information can be obtained from bug databases.



New Language Features – Place Holder
Analysis Tools – Efficient Race Detection
Run-time System – Place Holder
Code Development Tools – Synchronization Adjustment

Better understanding of synchronization issues faced by real-world developers is highly needed.

Overview

An empirical study on:

- Critical Section Evolution
- Over Synchronizations
- Concurrency Bug Origins

Difficulties Encountered

Large revision numbers(8 - 19 years)

- Apache - 26,000, Mozilla Firefox - 180,000, ...

Different kinds of synchronization primitives

- Different kinds of Mutex Locks, Condition Variables

Different kinds of synchronization primitive changes

- Critical Section Getting Larger, Smaller, Synchronization Type Change, Synchronization Variable Change etc.

Contribution

1. Comprehensive empirical study on how **lock-protected critical sections** are changed when software evolves.
2. Detailed case study on **over-synchronization** issues.
3. Detailed case study on **how concurrency bugs are introduced**.

Methodology for Critical Section Study

- 4 representative C/C++ open source projects
- Hierarchical taxonomy for all critical section changes
- Regular-expression based Python scripts for structural patterns
- Manual study on 500+ cases found by the scripts for purpose patterns

Software

4 representative C/C++ open-source software projects including **data base, web server, web browser, media player**

Applications	Period	# of Revisions	Latest Version Size(LoC)
Apache Web Server	1996 - 2014	25897	258K
Mozilla Browser	2007 - 2014	188000	8.17M
MPlayer Media Player	2001 - 2014	37000	448K
MySQL Database	2000 - 2014	6800	3.91M

Critical Section Change Structural Patterns

Adding critical sections:

- Add Synchronization

```
/* Empty the accept queue of completion contexts */  
+ apr_lock_acquire(qlock);  
while (qhead) {  
    CloseHandle(qhead->Overlapped.hEvent);  
    closesocket(qhead->accept_socket);  
    qhead = qhead->next;  
}  
+ apr_lock_release(qlock);
```

- Add All

```
+ pthread_mutex_lock(&rli->log_space_lock);  
+ rli->log_space_total -= rli->relay_log_pos;  
+ pthread_mutex_unlock(&rli->log_space_lock);
```

Critical Section Change Structural Patterns

Removing critical sections:

- Remove Synchronization

```
- pthread_mutex_lock(&THR_LOCK_keycache);  
+ /* pthread_mutex_lock(&THR_LOCK_keycache); */  
pthread_mutex_lock(&LOCK_status);  
for (i=0; variables[i].name; i++) {  
    ...  
    pthread_mutex_unlock(&LOCK_status);  
- pthread_mutex_unlock(&THR_LOCK_keycache);  
+ /* pthread_mutex_unlock(&THR_LOCK_keycache); */
```

- Remove All

```
    wait_for_refresh(thd);  
}  
- pthread_mutex_lock(&thd->mysys_var->mutex);  
- thd->mysys_var->current_mutex=0;  
- thd->mysys_var->current_cond=0;  
- pthread_mutex_unlock(&thd->mysys_var->mutex);  
return result;  
}
```

Critical Section Change Structural Patterns

Modifying existing critical sections

- Modify Body
- Modify Synchronization(Variable, Primitive, Boundary, Split, Unlock)
 - `VOID(pthread_mutex_lock(&hostname_cache->lock));`
 - + `VOID(pthread_mutex_lock(&LOCK_hostname));`
 - `if (!(hp=gethostbyaddr((char*) in,sizeof(*in), AF_INET))) {`
 - `VOID(pthread_mutex_unlock(&hostname_cache->lock));`
 - + `VOID(pthread_mutex_unlock(&LOCK_hostname));`
 - `DEBUG_PRINT("error","gethostbyaddr returned %d",errno); goto err; }`

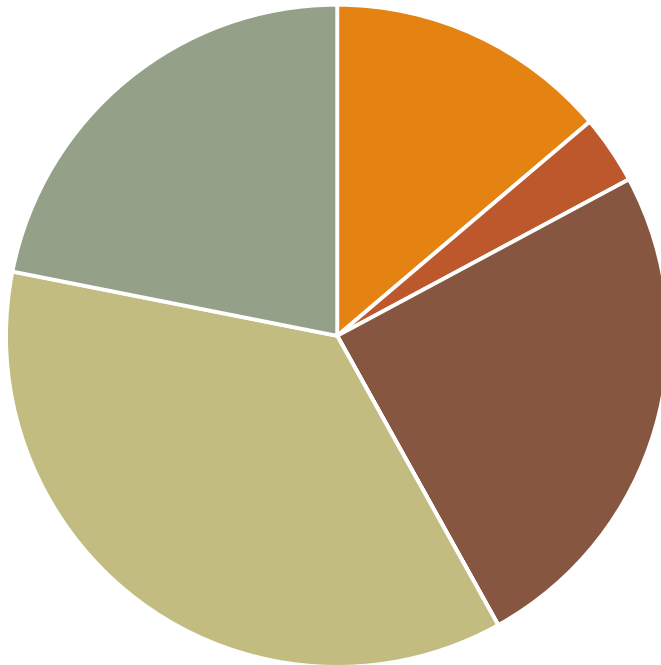
Observations

	Apache	Mozilla	MPlayer	MySQL
Add	138	227	65	1548
Rem	199	272	59	1411
Mod	467	204	33	4301
Total #	804	703	157	7260

	Add	Rem	Mod
Percentage	22%	21%	56%

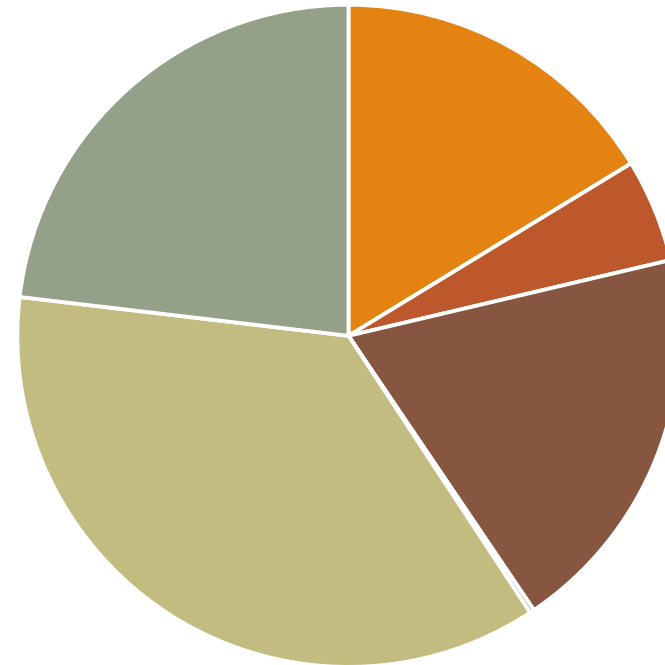
Observations

Apache



■ Add-all ■ Add-Sync ■ Rem-All ■ Rem-Sync ■ Mod-Body ■ Mod-Sync

MySQL



■ Add-all ■ Add-Sync ■ Rem-All ■ Rem-Sync ■ Mod-Body ■ Mod-Sync

Observations

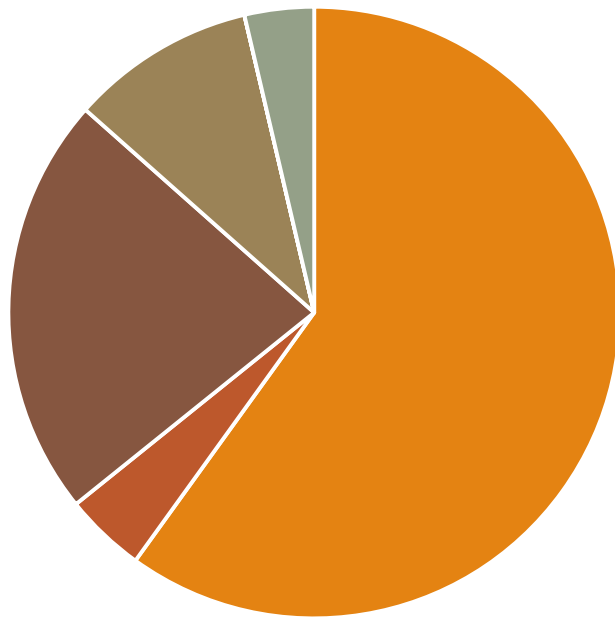
How many changes for **each critical section**?

	Apache	Mozilla	MPlayer	MySQL
Mod-Body	196	139	23	932
Mod-Sync	93	15	9	640
Total	261	149	23	1173

	Mod-Body	Mod-Sync
Percentage	80%	50%

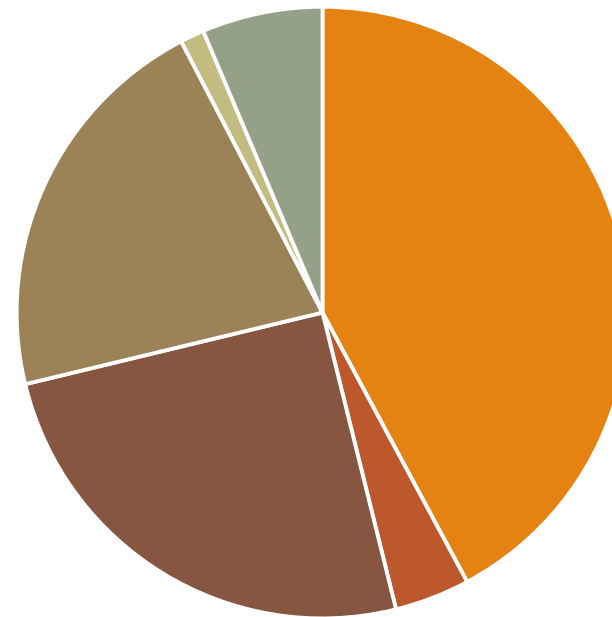
Observations

Apache



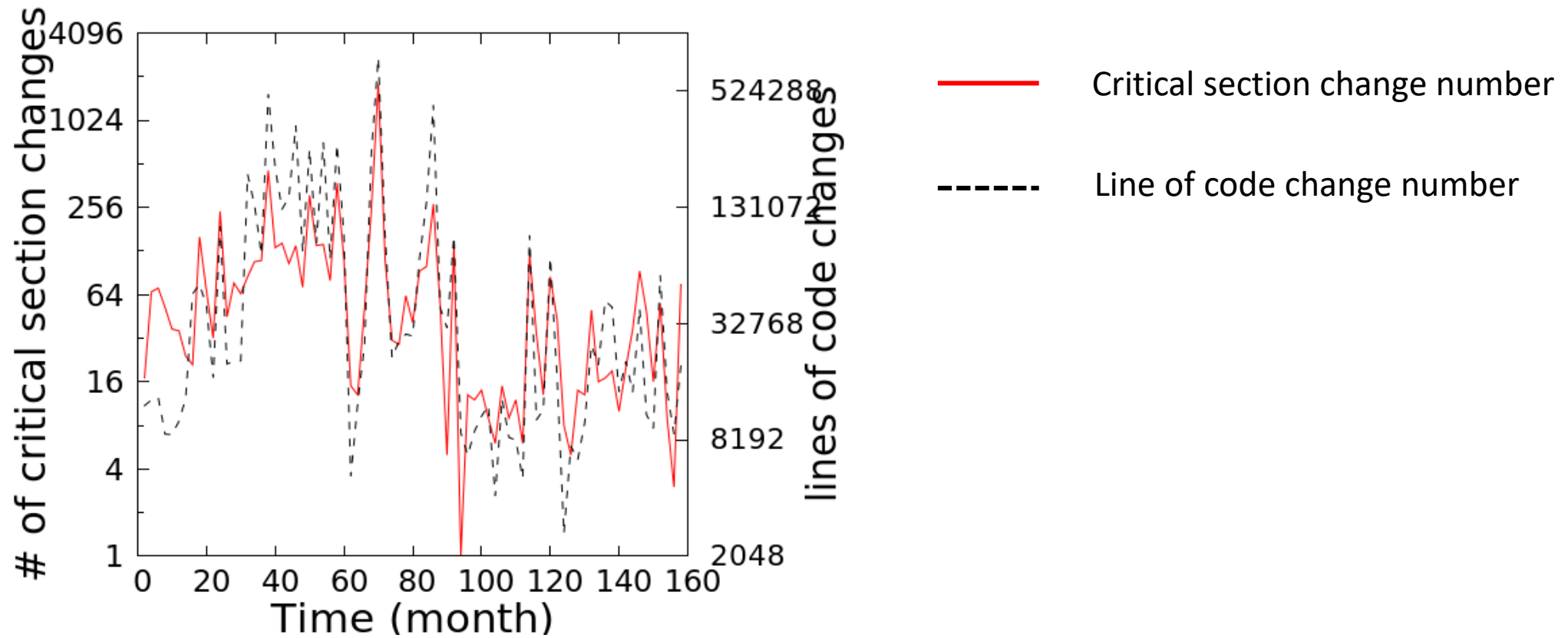
■ Mod-Body ■ Mod-Sync-Var ■ Mod-Sync-Primitive
■ Mod-Sync-Boundary ■ Mod-Sync-Split ■ Mod-Sync-Unlock

MySQL



■ Mod-Body ■ Mod-Sync-Var ■ Mod-Sync-Primitive
■ Mod-Sync-Boundary ■ Mod-Sync-Split ■ Mod-Sync-Unlock

Observations

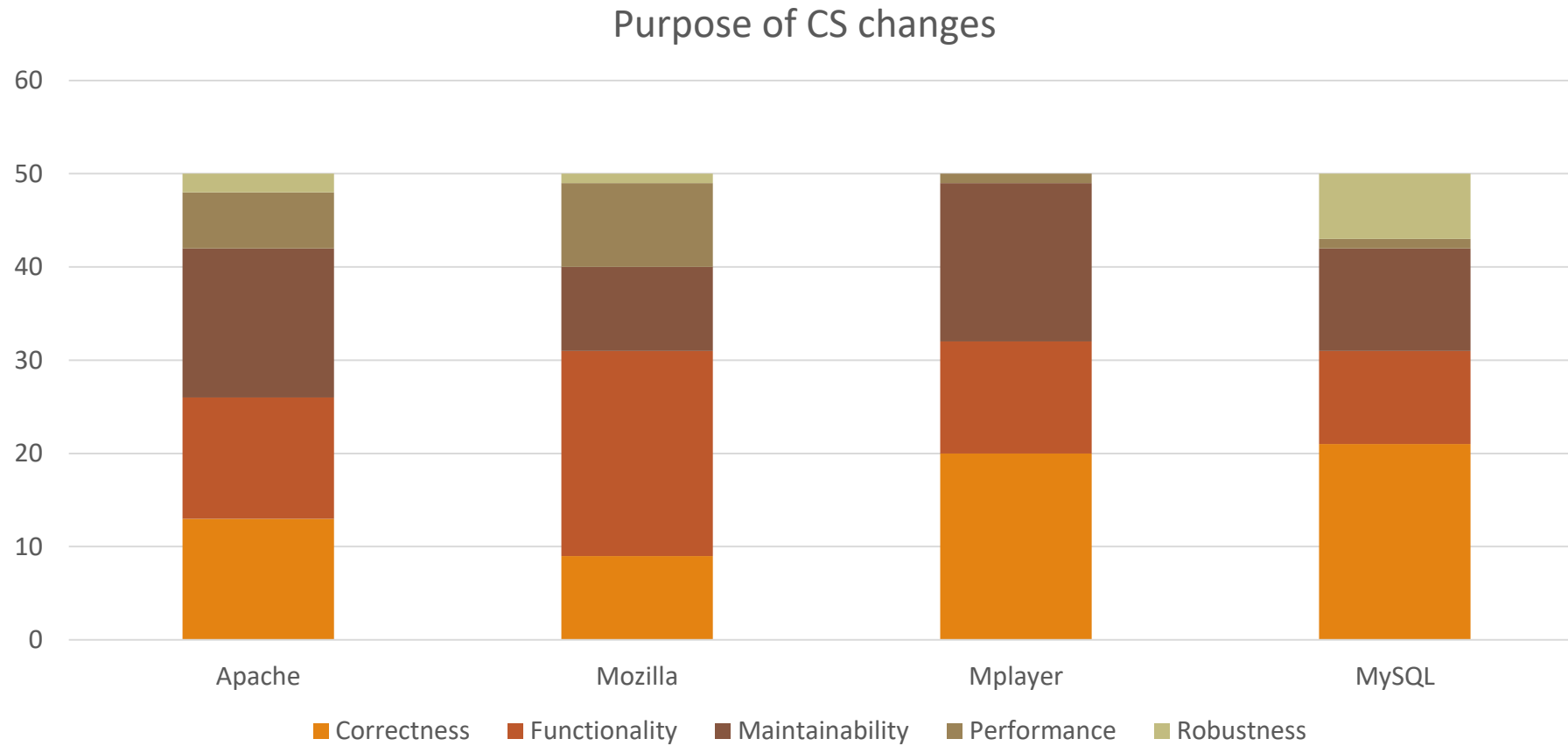


MySQL

Critical Section Change Purpose Patterns

Purpose	Patterns
Correctness	Fixing functional bugs
Functionality	Adding or changing code functionality
Maintainability	Code refactoring
Performance	Improving performance
Robustness	Adding sanity checks, logs.

Observations



Over Synchronization Study

- What is over synchronization?
 - Unnecessary synchronizations that will cause performance degradation
- Why do we care about it?
 - Performance considerations
 - Code maintenance considerations

Methodology and Threats to Validity

Randomly picked up 20 cases from Mod Sync Variable, Mod Sync Split, Mod Sync Boundary that fix/relieve over synchronization issues.

They do not cover all over-synchronization issues fixed by these three types of changes nor other over-synchronization issues fixed by other changes.

Observations

```
+static pthread_mutex_t LOCK_hostname;

...
    DEBUG_RETURN(0);    // out of memory
#else
- VOID(pthread_mutex_lock(&hostname_cache->lock));
+ VOID(pthread_mutex_lock(&LOCK_hostname));
    if (!(hp=gethostbyaddr((char*) in,sizeof(*in), AF_INET)))
    {
- VOID(pthread_mutex_unlock(&hostname_cache->lock));
+ VOID(pthread_mutex_unlock(&LOCK_hostname));
        DEBUG_PRINT("error",("gethostbyaddr returned %d",errno));
        goto err;
    }
```

Observations

Where to Apply the Changes?

- Critical sections with highly contended locks

How to Conduct Mod Sync Variable?

- Select a new lock
- Replace locks with existing ones
- Among 11 cases under study, original locks are replaced by new ones

Concurrency Bug Origins

Methodology and Threats to Validity

Manual studies on 40 bugs from two bug sources:

Bug Source One:

28 real-world concurrency bugs coming from more than 10 widely used C/C++ software projects, repeated and evaluated in 12 recent concurrency-bug related papers.

Bug Source Two:

12 randomly sampled concurrency bugs from our critical-section change study.

Taxonomy

Our categorization is based on three key ingredients:

1. Shared variables
2. Instructions that accessing these shared variables
3. Synchronization contexts(surrounding locks, preceding barriers)

Observations

Categorization of how concurrency bugs are introduced:

Type 1: One Thread, New Instructions in Old Context

```
/* Log Thread */
/*log status was OPEN*/
... //close old log
+ log_status = CLOSED;
... //open new log
log_status = OPEN;

/* Query Thread */
/*log after transaction*/
if(log_status==OPEN){
... // log update (#)
}else{
//Failure:transaction
// not logged
}
```

Figure 6: How the MySQL₇₉₁ bug was introduced

Type 2: One Thread, New Instructions in New Context

Observations

Categorization of how concurrency bugs are introduced:

Type 3: Multiple Threads, New Variables Accessed in Old Contexts

```
+ static httrackp *g_opt = NULL;

/*    Main Thread    */      /*    Child Thread    */
int main() {                  void child(...) {
    ...
    pthread_create(child, ...);    /*Initialize g_opt*/
+ mutexlock(&g_opt->s.l);          + g_opt = create_opt();
    ...
}                                  }
```

Figure 7: How HTTrack_{b20247} was introduced

Type 4: Multiple Threads, New Instructions in New Contexts

Discussion

Understanding how concurrency bugs are introduced can help us in:

- Improving the performance and accuracy of existing concurrency-bug analysis techniques
- Synchronization analysis
- Memory-access analysis
- Concurrency Bug Prevention

Conclusions
